Express Mail Label No. EL591789522US 07.24.00 A

UTILITY PATENT APPLICATION TRANSMITTAL
(Large Entity)

(Only for new nonprovisional applications under 37 CFR 1.53(b))

Docket No.
13675(YOR9-2000-0365US1)

Total Pages in this Submission
1

TO THE ASSISTANT COMMISSIONER FOR PATENTS
Box Patent Application
Washington, D.C. 20231

Transmitted herewith for filing under 35 U.S.C. 111(a) and 37 C.F.R. 1.53(b) is a new utility patent application for an invention entitled:

METHOD FOR COMPILING PROGRAM COMPONENTS IN A MIXED STATIC AND DYNAMIC ENVIRONMENT

and invented by:

Rajesh Bordawekar
Manish Gupta
Samuel Pratt Midkiff
Mauricio J. Serrano

If a CONTINUATION APPLICATION, *check appropriate box and supply the requisite information:*

☐ **Continuation** ☐ **Divisional** ☐ **Continuation-in-part (CIP)** of prior application No.: ________

Which is a:

☐ **Continuation** ☐ **Divisional** ☐ **Continuation-in-part (CIP)** of prior application No.: ________

Which is a:

☐ **Continuation** ☐ **Divisional** ☐ **Continuation-in-part (CIP)** of prior application No.: ________

Enclosed are:

Application Elements

1. ☒ Filing fee as calculated and transmitted as described below
2. ☒ Specification having 34 pages and including the following:
   - a. ☒ Descriptive Title of the Invention
   - b. ☐ Cross References to Related Applications *(if applicable)*
   - c. ☐ Statement Regarding Federally-sponsored Research/Development *(if applicable)*
   - d. ☐ Reference to Microfiche Appendix *(if applicable)*
   - e. ☒ Background of the Invention
   - f. ☒ Brief Summary of the Invention
   - g. ☒ Brief Description of the Drawings *(if drawings filed)*
   - h. ☒ Detailed Description
   - i. ☒ Claim(s) as Classified Below
   - j. ☒ Abstract of the Disclosure

UTILITY PATENT APPLICATION TRANSMITTAL
(Large Entity)

(Only for new nonprovisional applications under 37 CFR 1.53(b))

Docket No.
13675(YOR9-2000-0365US1)

Total Pages in this Submission
1

Application Elements (Continued)

3. ☒ Drawing(s) *(when necessary as prescribed by 35 USC 113)*
   - a. ☒ Formal Number of Sheets 11
   - b. ☐ Informal Number of Sheets ____

4. ☒ Oath or Declaration
   - a. ☒ Newly executed *(original or copy)* ☐ Unexecuted
   - b. ☐ Copy from a prior application (37 CFR 1.63(d)) *(for continuation/divisional application only)*
   - c. ☒ With Power of Attorney ☐ Without Power of Attorney
   - d. ☐ *DELETION OF INVENTOR(S)*
     Signed statement attached deleting inventor(s) named in the prior application, see 37 C.F.R. 1.63(d)(2) and 1.33(b).

5. ☐ Incorporation By Reference *(usable if Box 4b is checked)*
   The entire disclosure of the prior application, from which a copy of the oath or declaration is supplied under Box 4b, is considered as being part of the disclosure of the accompanying application and is hereby incorporated by reference therein.

6. ☐ Computer Program in Microfiche *(Appendix)*

7. ☐ Nucleotide and/or Amino Acid Sequence Submission *(if applicable, all must be included)*
   - a. ☐ Paper Copy
   - b. ☐ Computer Readable Copy *(identical to computer copy)*
   - c. ☐ Statement Verifying Identical Paper and Computer Readable Copy

Accompanying Application Parts

8. ☐ Assignment Papers *(cover sheet & document(s))*

9. ☐ 37 CFR 3.73(B) Statement *(when there is an assignee)*

10. ☐ English Translation Document *(if applicable)*

11. ☐ Information Disclosure Statement/PTO-1449 ☐ Copies of IDS Citations

12. ☐ Preliminary Amendment

13. ☒ Acknowledgment postcard

14. ☒ Certificate of Mailing
    - ☐ First Class ☒ Express Mail *(Specify Label No.)*: EL591789522US

UTILITY PATENT APPLICATION TRANSMITTAL (Large Entity) *(Only for new nonprovisional applications under 37 CFR 1.53(b))*	Docket No. 13675(YOR9-2000-0365US1)
	Total Pages in this Submission 1

Accompanying Application Parts (Continued)

15. ☐ Certified Copy of Priority Document(s) *(if foreign priority is claimed)*

16. ☐ Additional Enclosures *(please identify below)*:

Fee Calculation and Transmittal

CLAIMS AS FILED						
For	#Filed	#Allowed	#Extra		Rate	Fee
Total Claims	30	- 20 =	10	x	$18.00	$180.00
Indep. Claims	3	- 3 =	0	x	$78.00	$0.00
Multiple Dependent Claims (check if applicable) ☐						$0.00
					BASIC FEE	$690.00
OTHER FEE *(specify purpose)*						$0.00
					TOTAL FILING FEE	$870.00

☐ A check in the amount of   to cover the filing fee is enclosed.

☒ The Commissioner is hereby authorized to charge and credit Deposit Account No. **50-0510/IBM** as described below. A duplicate copy of this sheet is enclosed.

- ☒ Charge the amount of **$870.00** as filing fee.
- ☒ Credit any overpayment.
- ☒ Charge any additional filing fees required under 37 C.F.R. 1.16 and 1.17.
- ☐ Charge the issue fee set in 37 C.F.R. 1.18 at the mailing of the Notice of Allowance, pursuant to 37 C.F.R. 1.311(b).

Signature

Dated: **July 21, 2000**

Richard L. Catania
Registration No. 32,608

SCULLY, SCOTT, MURPHY & PRESSER
400 Garden City Plaza
Garden City, New York 11530
(516)742-4343

cc:

CERTIFICATE OF MAILING BY "EXPRESS MAIL" (37 CFR 1.10) Applicant(s): **Rajesh Bordawekar, et al.**		Docket No. **13675(YOR9-2000-0365US1)**	
Serial No. **Unassigned**	Filing Date **Herewith**	Examiner **Unassigned**	Group Art Unit **Unassigned**

Invention: **METHOD FOR COMPILING PROGRAM COMPONENTS IN A MIXED STATIC AND DYNAMIC ENVIRONMENT**

I hereby certify that the following correspondence:

New Utility Patent Application

(Identify type of correspondence)

is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 in an envelope addressed to: The Assistant Commissioner for Patents, Washington, D.C. 20231

July 21, 2000
(Date)

Mishelle Spina
(Typed or Printed Name of Person Mailing Correspondence)

(Signature of Person Mailing Correspondence)

EL591789522US
("Express Mail" Mailing Label Number)

Note: Each paper must have its own certificate of mailing.

METHOD FOR COMPILING PROGRAM COMPONENTS IN A MIXED STATIC AND DYNAMIC ENVIRONMENT

Background Of The Invention

Field of the Invention

The present invention relates to computer programming. More specifically, the invention relates to a method and variant of the method to compile programs or components of a program in a mixed static and dynamic environment.

Background Description

Most programming languages use the concept of a *data type* to identify a set of objects and operations that may be performed on those objects. Data types may be *primitive* (built into the language) or *user-defined*. A *class* in a programming language is used to create a user-defined type. A program written in an object-oriented manner can be viewed as a collection of classes. Classes contain declarations of both data and executable code in the form of *methods*. Herein, such methods are referred to as *procedures*.

Some programming languages, for example, Java®, have dynamic features like run-time binding of method calls and dynamic class loading. The term *virtual machine* is used herein to refer to the execution environment of such programming languages. Implementing a virtual machine for such a language may involve either interpreting the program or compiling it into the native code of the

target machine. Because interpretation incurs a high run-time overhead, virtual machines often rely on compilation for delivering high performance. Two prominent approaches to compilation in a virtual machine are dynamic compilation and static compilation. Dynamic compilation involves performing the translation of a program component (such as method or a collection of methods) to native machine code at run-time, before executing that program component. Static compilation involves performing the translation in an offline manner and generating one or more binary codes to be executed at run-time. Examples of virtual machines for Java using dynamic compilation include the IBM DK and the Sun JDK. Examples of static compilers for a Java-like language include JOVE, Tower Technologies TowerJ and the NaturalBridge BulletTrain compilers.

There are many problems with existing approaches to implementing virtual machines for dynamic languages. The problems with dynamic compilation include:

1. *Performance overhead of compilation at run-time*: The overhead of compilation is incurred every time the program is executed and is reflected in the overall execution time. Therefore, dynamic compilers tend to be less aggressive in applying optimizations that require deep analysis of the program.

2. *Testability and serviceability problems of the generated code:* Dynamic compilers that make use of run-time information about data characteristics to drive optimizations can lead to a different binary executable

being produced each time the program is executed. This can create reliability problems, as the code being executed may never have been tested.

3. *Large memory footprint:* A dynamic compiler is a complex software system with several interacting components, particularly if it supports aggressive optimizations. Hence, it usually has a large memory footprint, which gets directly added to the memory footprint of the application, since the dynamic compiler is invoked at run time. The memory footprint is particularly important for embedded systems, where the memory available on the device is limited.

Static compilation for dynamic languages leads to the following problems:

1. *Dynamic binding:* The code for dynamically linked class libraries may not be available during static compilation of a program, causing opportunities for interprocedural optimizations to be missed. Furthermore, the rules for binary compatibility in dynamic language like Java make it illegal to apply even simple inter-class optimizations -- e.g., method inlining across class boundaries -- unless the system has the ability to undo those optimizations in the event of changes to other classes.

2. *Dynamic class loading:* In general, dynamic class loading, as defined in languages like Java, requires the ability to handle a sequence of bytecodes representing a class (not seen earlier by the compiler) at run time. Hence, it is impossible for a virtual machine to support

a feature like dynamic class loading with a pure static compiler.

A digest of a data stream is a one-way hash function of the contents of the data stream that, with a very high probability, yields a different value if there are any changes made to the contents of the data stream. The Java 2 Security API supports secure hash functions to obtain the digest of a data stream or message.

Prior art for reducing the cost of dynamic compilation of Java can be found in An annotation-aware Java virtual machine implementation, *Proc. ACM SIGPLAN 1999 Java Grande Conference*, June 1999, A. Azevedo, A. Nicolau, and J. Hummel. The AJIT compiler annotates the byte-code with machine independent analysis information that allows the JIT to perform some optimizations without having to dynamically perform analysis. A serious limitation of this system is that program transformation and code generation still occur at application execution time.

Prior art for reducing the cost of dynamic compilation can be found in A general approach for run-time specialization and its application to C, *23rd ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 145-156, January 1996, C. Consel and F. Noel; An evaluation of staged run-time optimizations in DyC, *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999, B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers; and Dynamic specialization in the Fabius system, *ACM Computing Surveys*, September 1998, M. Leone and P. Lee.

DyC is a selective dynamic compilation system for C, which reduces the dynamic compilation overhead by statically preplanning the dynamic optimizations. Based on user annotations that identify variables with relatively few run-time values, it applies partial evaluation techniques to partition computations in regions affected by those variables into static and dynamic computations.

Other systems, such as Tempo (see, A general approach for run-time specialization and its application to C, *23rd ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 145-156, January 1996, C. Consel and F. Noel.) and Fabius (see, Dynamic specialization in the Fabius system, *ACM Computing Surveys*, September 1998,M. Leone and P. Lee.) support a similar staging of optimizations based on user annotations. All of these approaches have several limitations. First, they are unable to apply the staging of optimizations in the absence of user annotations. Second, they still require substantial code generation at run-time, and can only save the overhead of a few compiler optimizations. Third, they do not perform security checks to ensure the validity of code. Further, they do not deal with languages like Java, which have many dynamic features that make it difficult to generate code ahead of execution.

Prior art for recording the persistent execution state of a virtual machine for the Java platform can be found in Orthogonal persistence for the Java platform: Draft specification, October 1999,

http://www.sun.com/research/forest/index.html, M. Jordan and M. Atkinson; and Persistent execution state of a Java Virtual Machine, *Proc. ACM 2000 Java Grande Conference*, San Francisco, June 2000, T. Suezawa. These systems provide support for checkpointing the state of a Java application and virtual machine. They do not store the executable code for various procedures.

SUMMARY OF THE INVENTION

An object of this invention is to provide an improved method of compiling programs or components of a program in a mixed static and dynamic environment.

Another object of the present invention is to reduce the amount of time and memory spent in run-time compilation, while strictly honoring the semantics of dynamic features of a programming language.

A further object of this invention is to provide an improved method of compiling programs or components of a program in a mixed static and dynamic environment, so as to reduce the amount of time and resources spent in run-time compilations, and so as to exercise greater control over testing of the executable code for the program.

These and other objectives are attained with a method and system for a virtual machine in which compilation of a procedure is performed by (A) generating a persistent image, ahead of run time, that contains code for that procedure, and performing the following steps at run time; (B) checking for the existence and validity of a

code image for said procedure; (C) adapting the code image to the current execution context; and (D) using run-time compilation of the procedure if its code image does not exist, is invalid, or cannot be successfully adapted to the new execution context.

The preferred embodiment of this invention, as described below in detail, allows global interprocedural optimizations to be performed on the program, even if the programming language supports dynamic binding. Variants of the method show how one or several of the features of the method may be performed. The invention is particularly useful in the context of implementing Java Virtual Machines, although it can also be used in implementing other programming languages.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, aspects and advantages will be better understood from the following detailed description of a preferred embodiment of the invention with reference to the drawings, in which:

Figure 1 shows a block diagram of a prior art virtual machine in the context of which this invention may be used.

Figures 2 and 3 show a block diagram of a virtual machine using a method of this invention.

Figure 4 shows a block diagram of a QSI writer.

Figure 5 shows a block diagram of the dependence recorder.

Figure 6 shows pseudocode for the adaptation annotation recorder component.

Figure 7 shows a flow chart of the QSI recorder.

Figure 8 shows a flowchart of the QSI repository system.

Figures 9 and 10 shows a flow-chart of the QSRT compiler component.

Figure 11 shows a flowchart describing the validation checks performed on a QSI.

Figure 12 shows pseudocode for a method of adapting the code and auxiliary information for a procedure to a new execution context.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS OF THE INVENTION

Using a Mixed Static and Dynamic Environment

Figure 1 shows a prior art system, a virtual machine, to which this invention is applied. A computer program (100) is transformed into executable code (102) by the compiler (101). The compiler may either be invoked at run-time or in an offline manner. The executable code is run by a run-time system (103).

Figures 2 and 3 show a system using an embodiment of this invention. In the preferred embodiment, the compilation activity is broken up into two phases, described in

Figures 2 and 3. Referring now to Figure 2, the computer program (200) is processed by a quasi-static image generator compiler (201), referred to as *QSI writer*. The QSI writer produces one or more quasi-static images (202), referred to as *QSI*'s, which are persistent images of the executable code. The QSI's are stored for subsequent use by the virtual machine using a QSI repository system (203). Referring to Figure 3, the computer program (304), in the form of source code or intermediate language code, such as *bytecode* in a Java environment, is processed at run-time by a quasi-static run-time compiler (305), referred to as *QSRT compiler*. The QSRT compiler uses the QSI repository system (306) (which is identical to 203 in Figure 2) to retrieve the QSI's (307) containing executable codes for various components of the program. After processing the QSI, the QSRT compiler generates executable code (308) that is used by the run-time system (309) for executing the program.

Generation of Quasi-Static Images

Figure 4 shows a block diagram of a QSI writer. In the preferred embodiment of the method, the QSI writer is obtained by modifying a run-time compiler from prior art. In another embodiment, it is obtained by modifying a static, offline compiler. A front-end (401) processes the program to produce an intermediate code representation (402), which is fed to an optimizer (403) that produces optimized intermediate code (404). The front-end and optimizer represent well-known components of prior art compilers, and may be organized in different

ways, including, being organized in the form of multiple modules. The method of this invention adds to the optimizer a component (405) to record dependencies between different modules. This component is described further in Figure 5. The optimized intermediate code annotated with dependence information (406) produced by this component is processed by the back-end code generator (407), which may ignore the annotations on dependence information in the process of producing executable binary code (408). The method of this invention adds to the code generator an *adaptation annotation recorder* component (409), described further in Figure 6, which produces a further annotated executable code (410) with annotations to help adapt the code to a new execution context. The *QSI recorder* (411), described further in Figure 7 produces the QSI (412) which is stored for later processing.

The dependence recorder (405) from Figure 4 is described further in Figure 5. The *fine-grain dependence recorder* (500) keeps track of global optimizations performed by the compiler, and produces a list of fine-grain dependencies (501). In the preferred embodiment, these dependencies are recorded in the form of *class to procedure* dependencies. While compiling a procedure *A.foo* (i.e., a procedure *foo* of class *A*), for each optimization that exploits some information from a different class *B* (e.g., if a method *B.moo* is inlined into *A.foo*), the fine-grain dependence recorder in the method adds class *B* to the set of classes on which the code for *A.foo* is dependent. This allows the compiler, as explained later in the description of the QSRT

compiler, to perform dependence checks during program execution to avoid using stale code for a procedure in the event of changes to other code on which the code for that procedure is dependent.

The fine-grain dependencies (501) are processed by a *dependence granularity adjuster* (502) which replaces some fine-grain dependencies by coarser-grain dependencies to produce the final list of dependence annotations (503) to be used in the QSI. In the preferred embodiment, the dependence granularity adjuster examines the dependencies recorded for various procedures of each class. It factors out the dependencies that are common to all procedures in the class and records them at the *class to class* dependence level. The remaining dependencies continue to be recorded at the class to method level. For example, if a class A has two methods *foo* and *bar*, which are dependent, respectively, on classes B, C and B, D, the compiler would record the dependence of class A as a whole on B, and additionally, the dependence of *foo* on C and of *bar* on D.

The adaptation annotation recorder component (409) from Figure 4 is described further in the pseudocode shown in Figure 6. The list of annotations is initialized to be empty in Line 601. The loop in Line 602 processes each instruction in the machine code and each item recorded in the auxiliary information, such as exception tables and garbage collection maps in a language supporting exceptions and garbage collection. Line 603 checks if the current instruction or item is dependent on the

current execution context. If it is dependent, line 604 adds an annotation in the form of <I, T, S> to the list of adaptation annotations, where I is an identifier for the instruction or item, such as the offset of the instruction in the code, T is the type of this instruction or item (such as load of an instance field of an object), and S is a symbolic reference that expresses the execution context-dependent information in an execution context-independent manner, which allows the modified form of this instruction or item that is valid in a new execution context to be later generated. At the end of this procedure (Line 607), ANNOT contains a list of all annotations for adapting this code. To further explain how a compiler determines which instruction is dependent on an execution context and how it uses a symbolic reference to record the information in an execution context-independent manner, example is given below of adaptation annotations for a Java Virtual Machine code.

Consider a Java Virtual Machine implementation for the IBM PowerPC architecture in which references to all static fields and methods are represented by an index into a global table which holds all such references for different classes loaded by the program. In such a virtual machine, the offsets for fields and method references are determined by the order in which classes are loaded in a particular execution of a virtual machine. Due to lazy class loading, classes can be loaded in a different order in different virtual machine instances, thus requiring different values of these offsets in different virtual machine instances. Consider

an instruction in a method *foo* of class *bar* which loads the static field *stats.count* (field *count* of class *stats*). The following PowerPC load instruction is generated to access the field:

lwz R1=@{JTOC + offset of field *stats.count*

JTOC is a dedicated register pointing to the table of global variables, and offset of field stats.count is an immediate-signed field giving the position of stats.count in that table. The value of the offset is assigned when the class stats is loaded. The adaptation annotation for the instruction is <I, T, S>, where I is the offset of the lwz instruction, T is an identifier denoting static field access, and S is the symbolic reference to the *constant pool entry* (as defined in the Java language specification (see *The Java Language* Specification *(Java Series)*, James Gosling, Bill Joy and Guy L. Steele, Jr. Addison-Wesley Publishing Company, Reading, MA.)) in the class bar for *stats.count*. Due to procedure inlining across class boundaries, a procedure may contain references that do not appear in the constant pool of its defining class. The compiler in the preferred embodiment creates an *extended constant pool* to handle relocation for references that are imported from other classes. An extended constant pool entry consists of the pair <N, C >, where N is an index into the constant pool of the class C from which this reference has been imported.

It should be noted that a static field reference is used only as an example to illustrate how adaptation annotations are recorded. There are other kinds of instructions that need annotations for the purpose of adaptation. For example, loads and stores of instance

fields of objects and method references need annotations as well, since the offsets used for fields of objects and for virtual methods of a referenced class may change if its parent class (from which the referenced class is derived) changes between the time of writing and use of the quasi-static image. Furthermore, the resolution status of those referenced fields and methods may change in the different virtual machine instances. Those skilled in the art will recognize that in the context of other virtual machine implementations, there are many kinds of instructions and items of information such as exception tables and garbage collection maps, for which the compiler can easily identify an appropriate adaptation annotation.

A flow chart of the QSI recorder (411) from Figure 4 is shown in Figure 7. It is easier to understand the QSI recorder by also looking at Figure 8, which shows a layout of a QSI for a class used in the preferred method. In the preferred embodiment, the QSI recorder is invoked by the virtual machine just before the end of program execution meant for generating persistent code images. A class used during program execution is processed in 700, which examines each procedure declared in that class. Any procedure not compiled with a high optimization level is compiled with a high optimization level in 701. A QSI for the class is created in 702. Next, 703 stores information such as a predetermined constant value as magic number (identifying that this data structure is a QSI), the environment information such as virtual machine version, OS version, and the target architecture, in the header region of the QSI.

The time of creation of the QSI is recorded as a timestamp in 704. Any additional information needed to identify a loaded class, such as information about the *defining class loader* of the class (as defined in Java 2 specification *(see The Java Language* Specification *(Java Series)*, James Gosling, Bill Joy and Guy L. Steele, Jr., Addison-Wesley Publishing Company, Reading, MA.)) is also recorded in 704. In Java 2, a run-time class is uniquely identified by the pair <C,D >, where C is the fully qualified name of the class, and D is the *defining class loader* of that class (see Dynamic class loading in the Java virtual machine, S. Liang and G. Bracha. *Proc. 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, Vancouver, Canada, October 1998; and Sun Microsystems. Java 2 Platform, Standard Edition Documentation. *http://java.sun.com/docs/index.html.*). If the defining class loader, D, of a class, C, is the primordial class loader, no further information is recorded for the class. If, however, D is not the primordial class loader, information about D is recorded, in the QSI for C, as a *digest* of the classfile for D. This enables the virtual machine to check, during program execution, whether C was defined by the same class loader during offline compilation and execution. (The check for the primordial class loader being the same during the QSI generation and program execution is subsumed by the check for compatibility of virtual machine instances in these modes.)

In accordance with the layout of QSI shown in Figure 8, 705 leaves space for recording a digital signature for the QSI. The class to class dependence information, as obtained by the dependence recorder (405) described earlier, is written in 706, as a list of other classes on which the code being recorded in this QSI is dependent. A directory containing pointers to various procedure codes is created in 707. Note that a virtual machine may decide to create more than one code version for a given procedure, in order to perform optimizations based on *specialization* of procedures. The code for each procedure, along with auxiliary information such as exception tables, garbage collection maps, dependence information on other classes, and annotations for adaptation to a new execution context, is written to the QSI in 708.

A digest of the contents of the QSI is computed using a predetermined secure hashing function [see Proposed Federal Information Processing Standard for Secure Hash Standard. *Federal Register*, 57 (21), pages 3747-3749, January 1992). The digest is then encrypted using a well-known method (see *Applied Cryptography*: *Protocols, Algorithms, and Source Code in C,* B. Schneier, John Wiley and Sons, 1996.) to obtain a digital signature for the QSI, which is recorded at its predefined place in QSI in 709. The digital signature enables the virtual machine to detect any tampering of the QSI by a (malicious) user. Finally, 710 ensures that the above process (comprising steps 700 through 709) is continued for each class that is loaded during program execution.

A flowchart of the QSI repository system (306) from Figure 3 is shown in Figure 8. A QSI may be stored in a file or in the memory itself. The preferred embodiment uses a file. The first step (800) is to identify where to place the QSI for a class. The QSI may be logically viewed as a part of the file containing the code for class seen by the virtual machine. In the preferred embodiment, the QSI is stored in a separate file with a .qsi suffix, but the method keeps track of the association between each QSI file and the original file containing the class code. The QSI machine uses a definite mapping (in step 800) to determine the directory in which a QSI is placed, given a unique identification of the class.

How this mapping is used may be illustrated with an example from a Java virtual machine. The location in which a classfile is stored in Java can be viewed as having two components: the *repository* containing the class, and the directory structure implied by the *fully qualified name* of the class [8]. For example, a class MyPackage.Foo, appearing in a repository /vol/jdk/classes on an AIX platform, is stored in the directory /vol/jdk/classes/MyPackage. The repository containing a class is identified by its defining class loader (e.g., using a search based on the classpath environment variable). For each class loader, a fixed mapping is defined from the name of the repository holding the class to the repository holding the QSI file, should it exist. Consider a class loader that loads classes over the network. The preferred method would use a local repository for the QSI files. Within a repository, the

method uses the same directory structure for a QSI file as that implied by the fully qualified name of the class. In the above example (for the class Foo in /vol/jdk/classes/MyPackage), given a QSI repository mapping function that replaces the string ``classes'' by the string ``qsi'', the corresponding QSI will be stored as Foo.qsi in the directory /vol/jdk/qsi/MyPackage.

Returning to Figure 8, step 801 checks for the existence of a QSI for the given class in the directory identified in step 800. A write request for a QSI is further processed using steps 802 through 803, while a read request for a QSI is processed using steps 804 through 805. Step 802 checks if the existing QSI should be modified in response to the write request. In the preferred embodiment, this checks the timestamp of the existing QSI. If it finds that the QSI is up-to-date, i.e., more recent than the file holding the class code, it decides not to overwrite the QSI. If the QSI is not up-to-date, then it deletes the older QSI, and the system proceeds to writing the new QSI in step 803. It should be noted that even when compiling a program for the first time, a QSI for a class from a library that is shared with other programs may already exist. Step 804 is followed for a read request if Step 801 shows that a QSI exists - it simply returns the QSI file. If for a read request, 801 shows that a QSI does not exist, a null value is returned by the read request in 805.

Program Execution: Reuse of Quasi-Static Images

Figures 9 and 10 show a flow-chart of the QSRT compiler component (305) shown in Figure 3. The QSRT compiler is obtained by modifying a prior art run-time optimizing compiler in accordance with the methods of this invention. A controller (900) in the run-time compiler makes decisions on when a procedure is to be compiled. Step 901 checks if a QSI for the declaring class of that procedure has been already loaded. If it has not been loaded, 902 checks for the existence of a QSI for that class in the system, using a read request of the QSI repository system (306). If a non-null QSI is returned by the step, 903 performs validation and security checks on the QSI to determine if the QSI can safely be used. This step is described further in Figure 11. Step 904 reads the dependence list for the class stored in the QSI and performs dependence checks to see if any of the other classes, on which the code for the given class is dependent, have changed. This is done by comparing the timestamps of files holding the codes for classes in the dependence list with the timestamp of the given QSI. If any of those files have a more recent timestamp than the timestamp of the QSI, the dependence check fails.

Such dependence checking allows the virtual machine to ensure the validity of generated code while performing global optimizations like inlining across class boundaries, in the presence of changes to other codes. If the dependence check passes, 905 reads the method directory area in the QSI to look up the pointer to the code for the procedure to be compiled and reads that code

from the QSI. If this code is found, 906 performs dependence checks, again using timestamps, to see if any of the classes on which code for this method is dependent have changed since the time the method code was generated. If this check passes, 907 adapts the code and auxiliary information for the procedure to the new execution context. This step is further described in Figure 12. If the adaptation of code succeeds, 908 returns the adapted executable code as the compiled code for the procedure. If any of the previous steps 902 through 907 fail, as shown in the flowchart, 909 performs run-time compilation of the procedure.

Figure 11 shows a flowchart providing further details of step 903 in Figure 9. Step 1100 reads the header data from the QSI. Step 1101 performs a compatibility check to ensure that the recorded QSI has been produced under an environment compatible with the environment of the current virtual machine instance. This involves verifying the magic number recorded in the QSI, checking the virtual machine version, operating system version, and the target architecture identifier, and ensuring that those are compatible with the current virtual machine, operating system, and target architecture. Step 1102 performs an out-of-date check, using timestamps, to see if the QSI is older than the file holding code for the corresponding class. In a virtual machine for Java 2, this test includes an additional check to ensure that the defining class loader (if it is not the primordial class loader) for the class being compiled is identical to the defining class loader for the class at the time of QSI creation.

This is done by computing a digest of the classfile for the defining class loader class (if it is not the primordial class loader) and comparing it with the recorded digest of defining class loader in the QSI. It may be noted that if the defining class loader is the primordial class loader, this check is subsumed by the check in 1101 for compatibility between virtual machine versions. A security check on the QSI is performed in 1103. This involves computing a digest of the QSI using the predetermined secure hash function. By comparing the encrypted form of this digest with the pre-recorded digital signature, this step verifies whether the QSI is bona-fide. In a Java virtual machine, since the binary code stored in the QSI is produced by the JVM only after the original code passed Java verification, this signature certifies that no further Java verification is needed for this code. Thus, it prevents a malicious user from using the QSI to bypass the safety features of Java.

Now described are further details of step 907 from Figure 10 to adapt the code and auxiliary information for a procedure to the new execution context. Figure 12 shows a pseudocode for this step. The loop 1201 goes over every item recorded in the list of adaptation annotations. Step 1202 locates the instruction in the code or the item in the auxiliary information for code, such as exception table or garbage collection maps, which is to be modified. Step 1203 locates the old information that is no longer relevant in the new execution context, and replaces it by new information computed using the symbolic reference S and information available to the

virtual machine about the new execution context. If this step fails, 1204 exits the adaptation procedure with an indication of failure. If step 1203 involved adding extra instructions to the code, step 1205 updates auxiliary information for the code, such as garbage collection maps and exception table, if necessary. For example, in the context of Java, if the original instruction is in a *try* block, the new instructions inserted as a replacement of the original instruction are also in that *try* block, thus we need to modify or add an entry in the exception table. To illustrate this procedure, once again we use an example from a Java Virtual Machine implementation for the IBM PowerPC architecture, discussed earlier while explaining the process of generating adaptation annotations.

Consider the sample instruction (discussed earlier) in *foo.bar* which loads the static field *stats.count*, leading to the following PowerPC load instruction being generated in the QSI to access the field:

lwz R1=@{JTOC + offset of field *stats.count*

JTOC is a dedicated register pointing to the table of global variables, and offset of field stats.count is an immediate-signed field giving the position of stats.count in the table in the virtual machine instance in which the QSI was generated. The adaptation annotation for the instruction is <I, T, S>, where I is the offset of the lwz instruction, T is an identifier denoting static field access, and S is the symbolic reference to the *constant pool entry* (as defined in the Java language specification (see *The Java Language* Specification *(Java Series)*, James Gosling, Bill Joy and Guy L. Steele, Jr. Addison-Wesley

Publishing Company, Reading, MA.) in the class bar for *stats.count*. Note that the class *foo* is already loaded and resolved before the virtual machine controller decides to compile *bar*. Therefore, the virtual machine has information mapping the entry S in the constant pool of class *foo* to a new entry in the table being used to hold static fields. The old offset for *stats.count* is replaced by the offset of this new entry.

Those skilled in the art will recognize that if S was referring to an entry in the *extended constant pool*, denoting an entry from the constant pool of another class whose method was inlined into *bar*, the adaptation could similarly be done using the mapping of constant pool for that class. It should be further noted that in some situations, extra code has to be inserted to do the adaptation

In the above example, if the class *stats* has not been loaded and resolved at the time of adaptation of the code for *foo.bar* (whereas the class was loaded and resolved when the QSI for *foo* was generated), the virtual machine would generate code of the following form:

```
label:
      lwz   R1=@{JTOC + offset of field table (1)
      lwz   R1=@{R1 + field Id               (2)
      if (R1 == 0) goto resolve               (3)
      lwz   R1=@{JTOC + R1 // field access    (4)
      ...

   resolve:
      (resolve field)                         (5)
```

```
b      label      // goto label                (6)
```

The first instruction loads the offset of a table used for resolving fields. The second instruction loads an entry from that table using a unique field identification number as an index. The third instruction tests if the entry is zero (the value for unresolved fields), and the fourth instruction performs the field access. The code for field resolution is placed in line (5), but for brevity is not shown in this example.

Note that a static field reference has been used only as an example to illustrate how adaptation annotations are recorded. Those skilled in the art will recognize that in the context of other virtual machine implementations there are many kinds of instructions and items of information such as exception tables and garbage collection maps, for which the compiler can easily identify an appropriate adaptation annotation.

Variants of Method

In addition to the preferred embodiment as described above, various modifications are now described.

In an alternate embodiment of the method, a single quasi-static image (QSI) is created for a collection of classes rather than a separate QSI for each class. For example, in Java, a single QSI may be created for a *package*. With another embodiment, more than one QSI may be created for each class, for example, this embodiment will create a separate QSI for each procedure in the class.

In other alternate embodiments of the method, the generation of QSI files is not necessarily done in a separate phase from the execution of the program. In one such embodiment, the step 909 performing run-time compilation of a procedure is followed by addition of the newly generated code along with auxiliary information to an existing or new QSI for the class containing the procedure. This allows the QSI's for a code to evolve with time in response to adaptive compilation. However, this embodiment can lead to additional overhead at run-time to create the QSI.

An alternate embodiment uses digests of classfiles rather than using timestamps for out-of-date checks in step 1102. In this embodiment, the virtual machine records a digest of the original classfile and of the classfiles on the dependence list, in the QSI file. The out-of-date check is performed by comparing the digest of the current classfile with the recorded digest for that file. An advantage of this approach is that trivial changes to a classfile's timestamp (due to the file being *touched* or moved) do not cause an unnecessary invalidation of the quasi-static image file.

Yet another alternate embodiment performs the reading of procedure code from the QSI file in an *eager* manner rather than in a *lazy* manner in the QSRT compiler. Rather than reading the code and auxiliary information for a procedure from a QSI only when responding to a virtual machine request to compile a procedure (in Step 905), this information is read, in this embodiment, when

the QSI file is read for the first time, which in turn happens the first time that compilation is attempted of *any* method in that class. This is useful when most (or all) procedures stored in the QSI are used during program execution, because sequential I/O is more efficient due to buffer prefetching. On the other hand, this has a potential drawback of leading to unnecessary I/O if relatively few procedures stored in the QSI are needed during program execution.

An alternate embodiment allows multiple code versions of a procedure to appear in a single QSI. Therefore, library code may be specialized for different applications. Furthermore, since compilation is done in an offline manner, the compiler has more freedom to apply potentially expensive interprocedural analysis to discover opportunities for specialization.

An alternate embodiment uses a different strategy for recording dependence information in Step 405 to explore the trade-offs between the overhead of dependence checking and the likelihood of QSI invalidation during program execution time. In this embodiment, the compiler moves a dependence item shared by the important (but not all) procedures to a class-level dependence, to reduce the overhead of dependence checking. This comes at the expense of possibly invalidating the entire QSI file rather than invalidating just the quasi-static code for a single procedure. In yet another embodiment, the compiler keeps dependence information at a finer granularity (e.g., procedure-to-procedure dependence) to reduce the

chances of invalidating a QSI, at the expense of increased overhead of dependence checking.

Another embodiment of the method does not use run-time compilation in step 909, in order to handle procedures for which executable code could not be not obtained using a QSI. Rather than run-time compilation, the virtual machine uses interpretation of such a procedure. An advantage of this approach is that it leads to a smaller memory footprint at run-time, which can be particularly useful for embedded and hand-held devices.

Thus, it should be understood that the preferred embodiment is provided as an example and not as a limitation. While the invention has been described in terms of a single preferred embodiment, with several variants, those skilled in the art will recognize that the invention can be practiced with modification within the spirit and scope of the appended claims.

CLAIMS

Having thus described our invention, what we claim as new and desire to secure by Letters Patent is as follows:

1. A method for a virtual machine in which compilation of a procedure is performed by:

A) generating a persistent image, ahead of run time, that contains code for that procedure, and performing the following steps at run time.

B) checking for the existence and validity of a code image for said procedure.

C) adapting the code image to the current execution context.

D) using run-time compilation of the procedure if its code image does not exist, is invalid, or cannot be successfully adapted to the new execution context.

2. Method of claim 1, where the virtual machine is a Java Virtual Machine.

3. Method of claim 1, where the persistent image in step A is generated by a run-time compiler during a prior execution of a program containing said procedure.

4. Method of claim 1, where the persistent image in step A is generated by a static compiler.

5. Method of claim 3, where any method not compiled by the run-time compiler during said prior execution is compiled in an offline manner in order to produce its persistent code image.

6. Method of claim 1, where the validation check in step B includes checking the virtual machine version recorded in the code image.

7. Method of claim 1, where the validation check in step B includes checking the operating system version recorded in the code image.

8. Method of claim 1, where the validation check in step B includes checking the target architecture identification recorded in the code image.

9. Method of claim 1, where the validation check in step B includes checking the timestamp of the generated code in the code image.

10. Method of claim 1, where the validation check in step B includes checking the timestamp of the said procedure's source code or intermediate code being compiled.

11. Method of claim 1, where the validation check in step B includes checking a digest of the generated code for said procedure in the code image.

12. Method of claim 1, where the validation check in step B includes checking the timestamp of one or more

procedures' source code or intermediate code on which the code for the said procedure is dependent.

13. Method of claim 1, where the validation check in step B includes checking a digest of one or more procedures' source code or compiled code on which the code for the said procedure is dependent.

14. Method of claim 1, where:

step A is further modified to generate code annotations which identify instructions that are dependent on the current execution context and which allow parameters valid for the new execution context to be deduced, and

adaptation of code image in step C is performed by using the annotations recorded by the modified step A described above.

15. Method of claim 14, adaptation of code image in step C involves adding extra instructions into the said code.

16. Method of claim 1, where step D uses interpretation instead of run-time compilation of the procedure if its code image does not exist, is invalid, or cannot be successfully adapted to the new execution context.

17. Method of claim 1, where the persistent code image is stored in a file.

18. Method of claim 1, where the persistent code image is stored in a memory device of the computer.

19. Method of claim 1, where a single persistent code image contains code for one or more procedures declared in a class in the user program or library.

20. Method of claim 2, where a single persistent code image contains code for a single Java class.

21. Method of claim 2, where a single persistent code image contains code for a single Java package.

22. Method of claim 1, where the code for all procedures stored in a persistent code image is read at run-time when any procedure code resident in that image is first read.

23. A system for operating a virtual machine in which compilation of a procedure is performed by:

A) means for generating a persistent image, ahead of run time, that contains code for that procedure;

B) means for checking, at run time, for the existence and validity of a code image for said procedure;

C) means for adapting, at run time, the code image to the current execution context; and

D) means for using run-time compilation of the procedure if its code image does not exist, is invalid, or cannot be successfully adapted to the new execution context.

24. A system according to Claim 23, where the virtual machine is a Java Virtual Machine.

25. A system according to Claim 23, where the means for generating the persistent image is a run-time compiler during a prior execution of a program containing said procedure.

26. A system according to Claim 23, where the means for generating the persistent image is a static compiler.

27. A program storage device readable by a machine, tangibly embodying a program of instructions executable by the machine to perform method steps for a virtual machine in which compilation of a procedure is performed, said method steps comprising:

A) generating a persistent image, ahead of run time, that contains code for that procedure, and performing the following steps at run time.

B) checking for the existence and validity of a code image for said procedure.

C) adapting the code image to the current execution context.

D) using run-time compilation of the procedure if its code image does not exist, is invalid, or cannot be successfully adapted to the new execution context.

28. A program storage device according to Claim 27, where the persistent image in step A is generated by a run-time compiler during a prior execution of a program containing said procedure.

29. A program storage device according to Claim 27, where the persistent image in step A is generated by a static compiler.

30. A program storage device according to Claim 27, where the virtual machine is a Java Virtual Machine.

METHOD FOR COMPILING PROGRAM COMPONENTS IN A MIXED STATIC AND DYNAMIC ENVIRONMENT

ABSTRACT OF THE DISCLOSURE

This invention describes a method and several variants for compiling programs or components of programs in a mixed static and dynamic environment, so as to reduce the amount of time and memory spent in run-time compilation, or to exercise greater control over testing of the executable code for the program, or both. The invention involves generating persistent code images prior to program execution based on static compilation or dynamic compilation from a previous run, and then, adapting those images during program execution. We describe a method for generating auxiliary information in addition to the executable code that is recorded in the persistent code image. Further, we describe a method for checking the validity of those code images, adapting those images to the new execution context, and generating new executable code to respond to dynamic events, during program execution. Our method allows global interprocedural optimizations to be performed on the program, even if the programming language supports, or requires, dynamic binding. Variants of the method show how one or several of the features of the method may be performed.The invention is particularly useful in the context of implementing Java Virtual Machines, although it can also be used in implementing other programming languages.

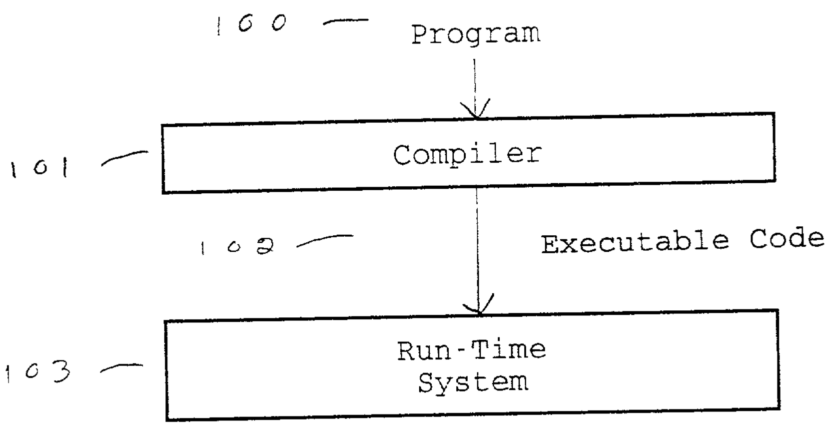
100
Program
101
Compiler
102
Executable Code
103
Run-Time
System

Fig. 1

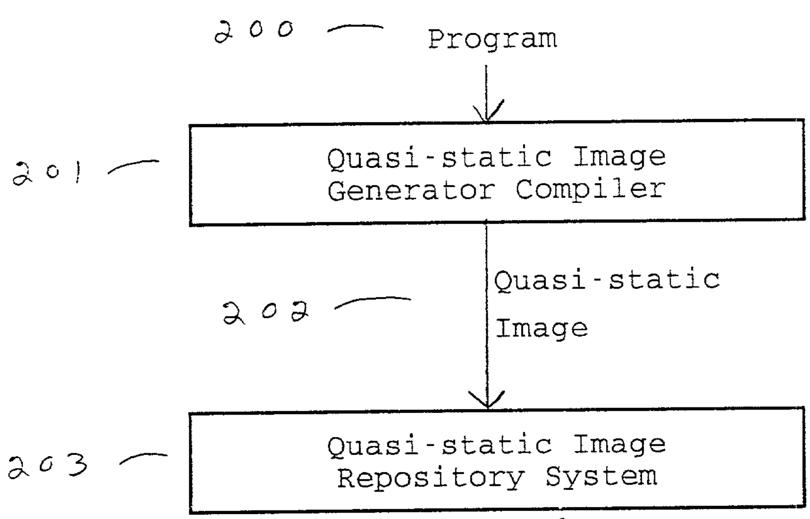
200
Program
201
Quasi-static Image Generator Compiler
202
Quasi-static Image
203
Quasi-static Image Repository System

Fig. 2

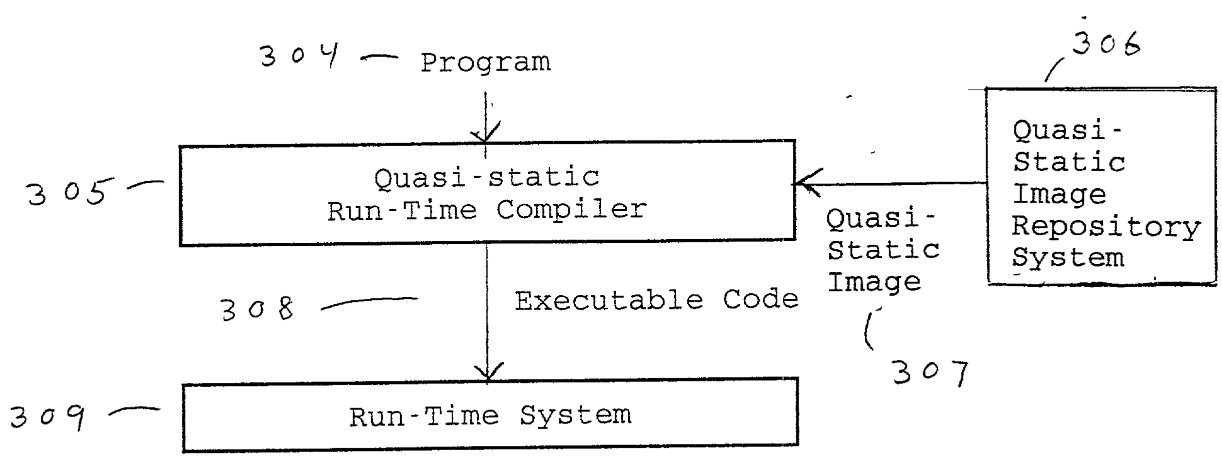
304
Program
305
Quasi-static Run-Time Compiler
306
Quasi-Static Image Repository System
Quasi-Static Image
307
308
Executable Code
309
Run-Time System

Fig. 3

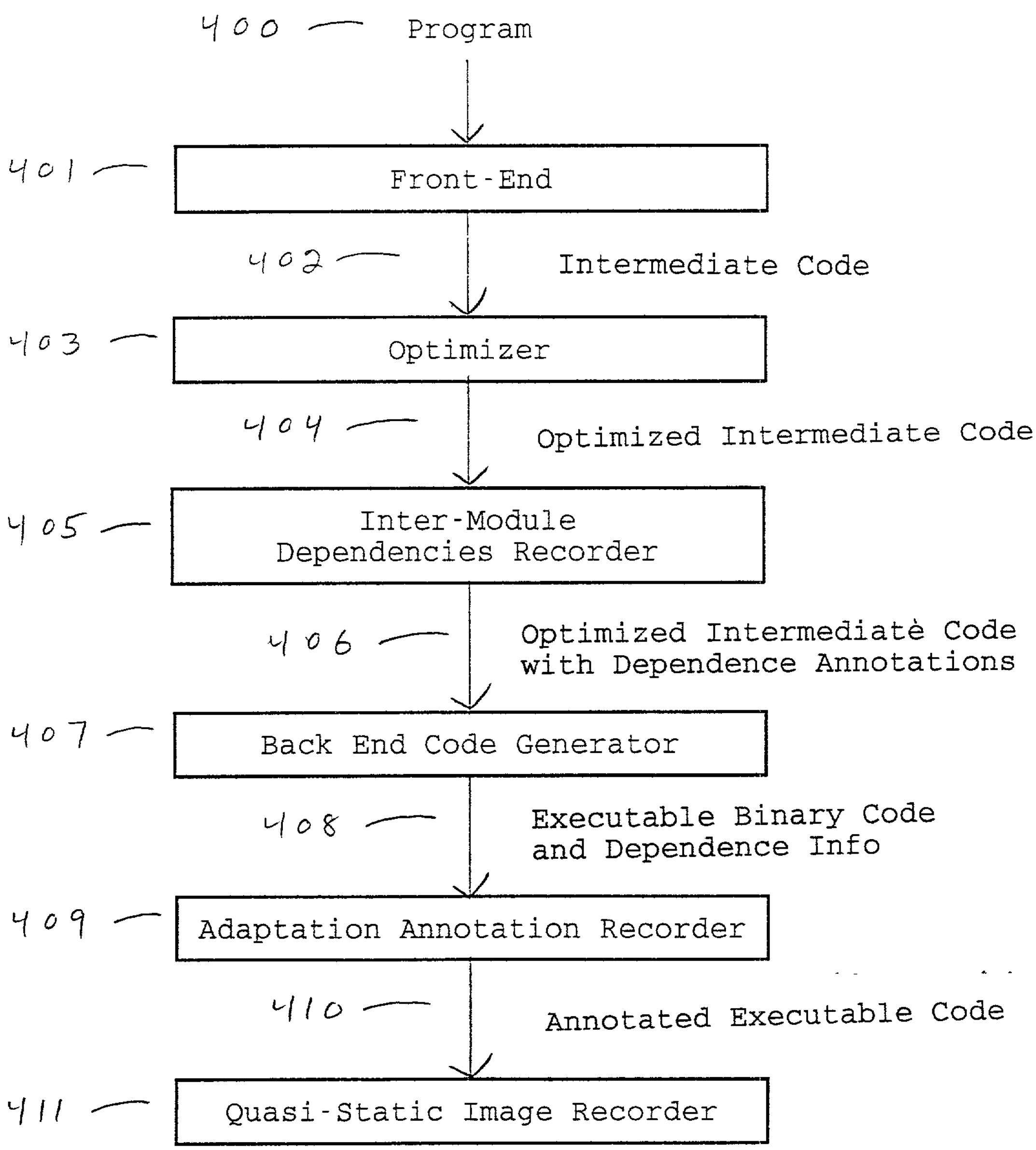
400
Program
401
Front-End
402
Intermediate Code
403
Optimizer
404
Optimized Intermediate Code
405
Inter-Module
Dependencies Recorder
406
Optimized Intermediate Code
with Dependence Annotations
407
Back End Code Generator
408
Executable Binary Code
and Dependence Info
409
Adaptation Annotation Recorder
410
Annotated Executable Code
411
Quasi-Static Image Recorder

Fig. 4

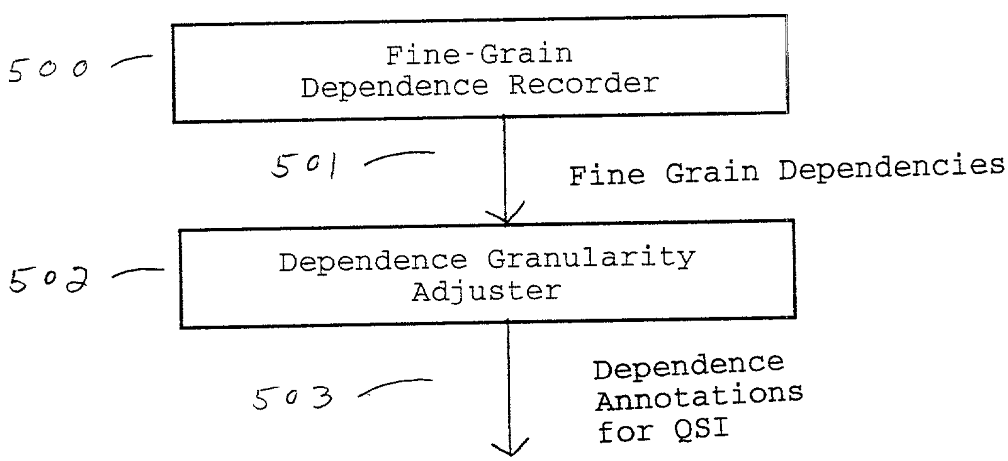
500
Fine-Grain
Dependence Recorder
501
Fine Grain Dependencies
502
Dependence Granularity
Adjuster
503
Dependence
Annotations
for QSI

Fig. 5

```
AdaptationAnnotationRecorder(code, auxiliary_info) {
  list ANNOT = empty;
  for each I which is an instruction in code or item in auxiliary_info do {
      if (I is dependent on current execution context) {
      add <I, instruction_type, symbolic_reference> to the list ANNOT;
      }
  }
}
```

Figure 6: Adaptation annotation recorder.

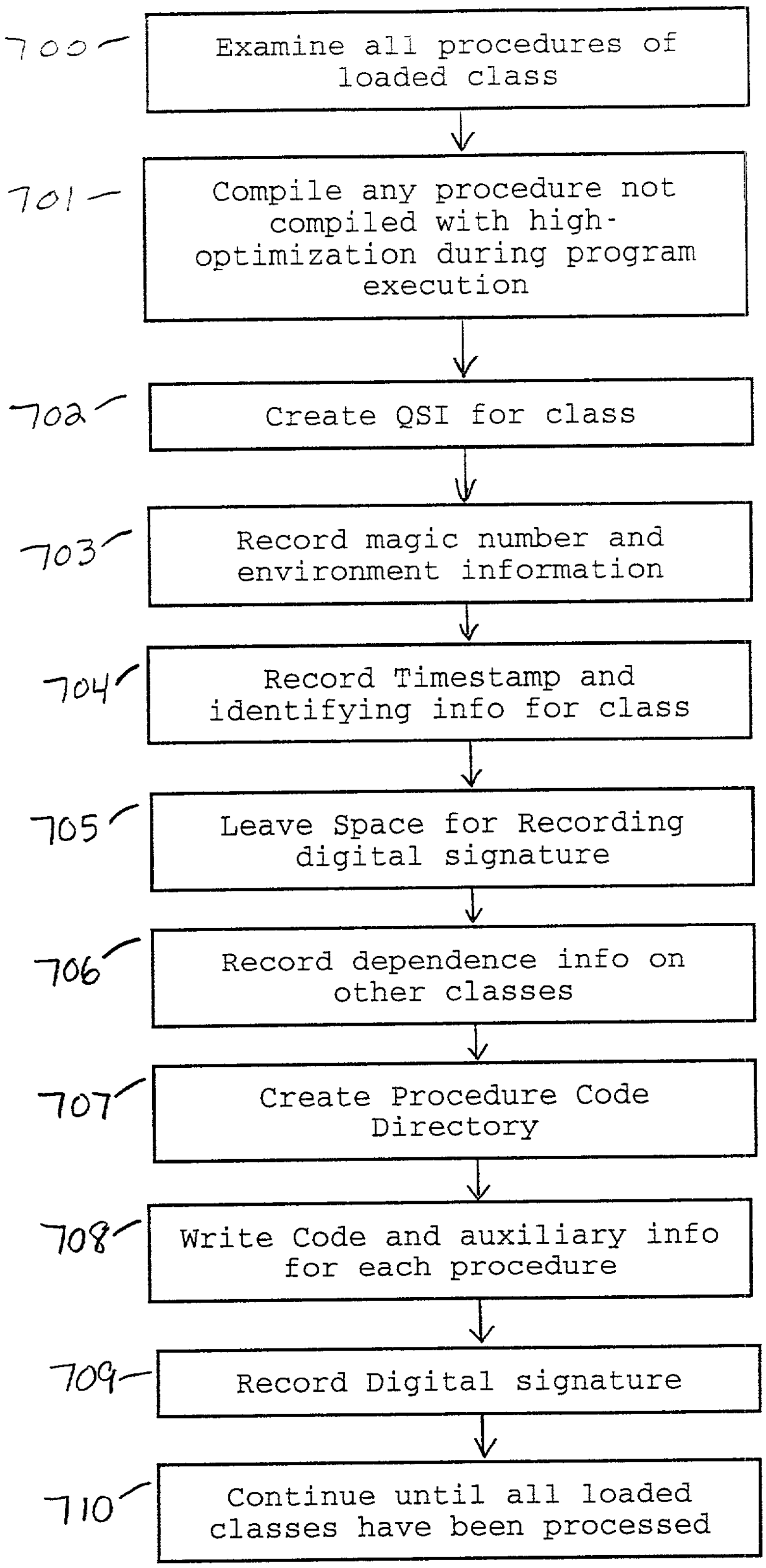
700
Examine all procedures of loaded class
701
Compile any procedure not compiled with high-optimization during program execution
702
Create QSI for class
703
Record magic number and environment information
704
Record Timestamp and identifying info for class
705
Leave Space for Recording digital signature
706
Record dependence info on other classes
707
Create Procedure Code Directory
708
Write Code and auxiliary info for each procedure
709
Record Digital signature
710
Continue until all loaded classes have been processed

Fig. 7

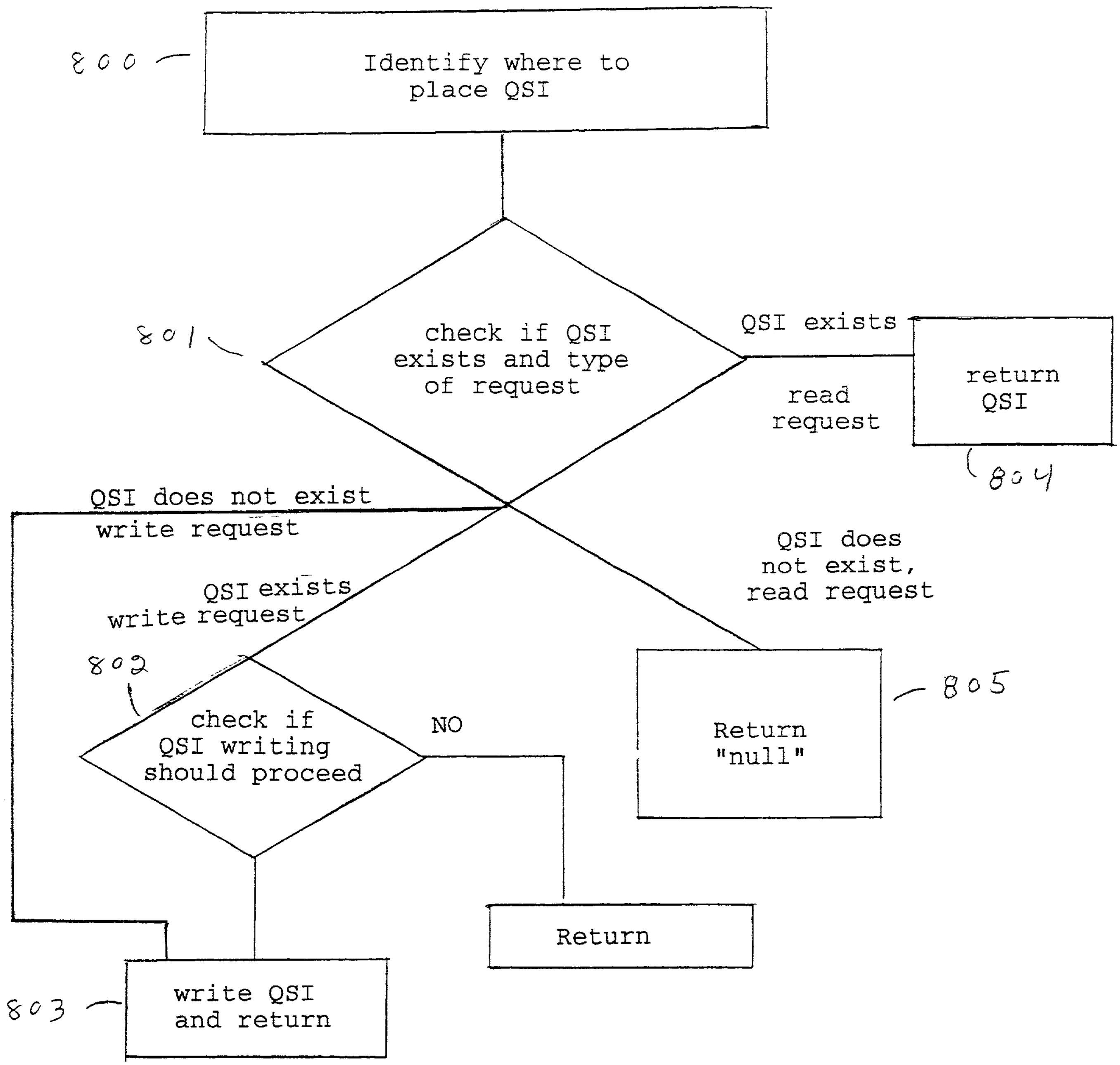
800
Identify where to place QSI
801
check if QSI exists and type of request
QSI exists
read request
return QSI
804
QSI does not exist
write request
QSI does not exist, read request
QSI exists
write request
802
check if QSI writing should proceed
NO
Return "null"
805
Return
803
write QSI and return

Fig. 8

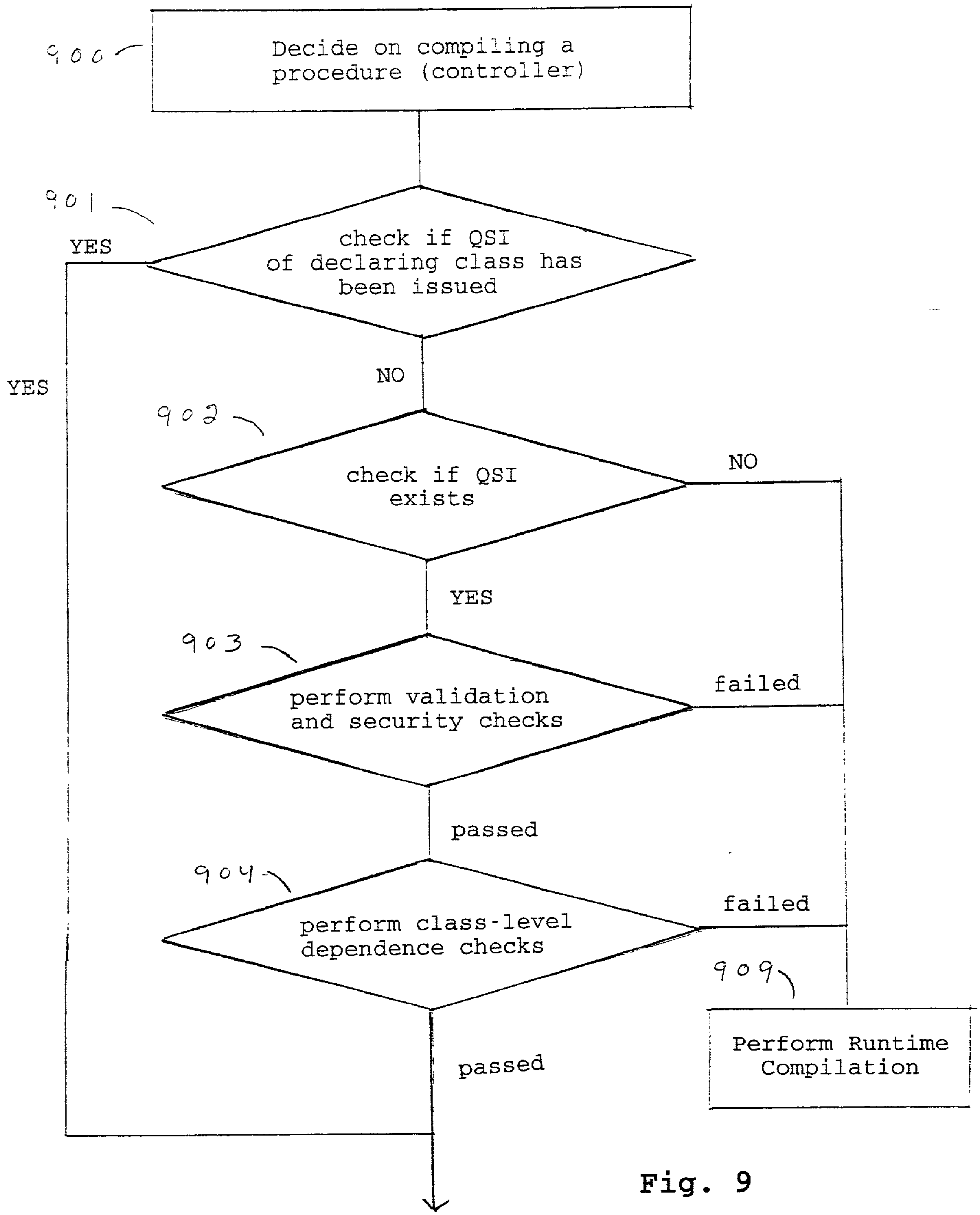
900
Decide on compiling a procedure (controller)
901
YES
check if QSI of declaring class has been issued
YES
NO
902
check if QSI exists
NO
YES
903
perform validation and security checks
failed
passed
904
perform class-level dependence checks
failed
909
Perform Runtime Compilation
passed

Fig. 9

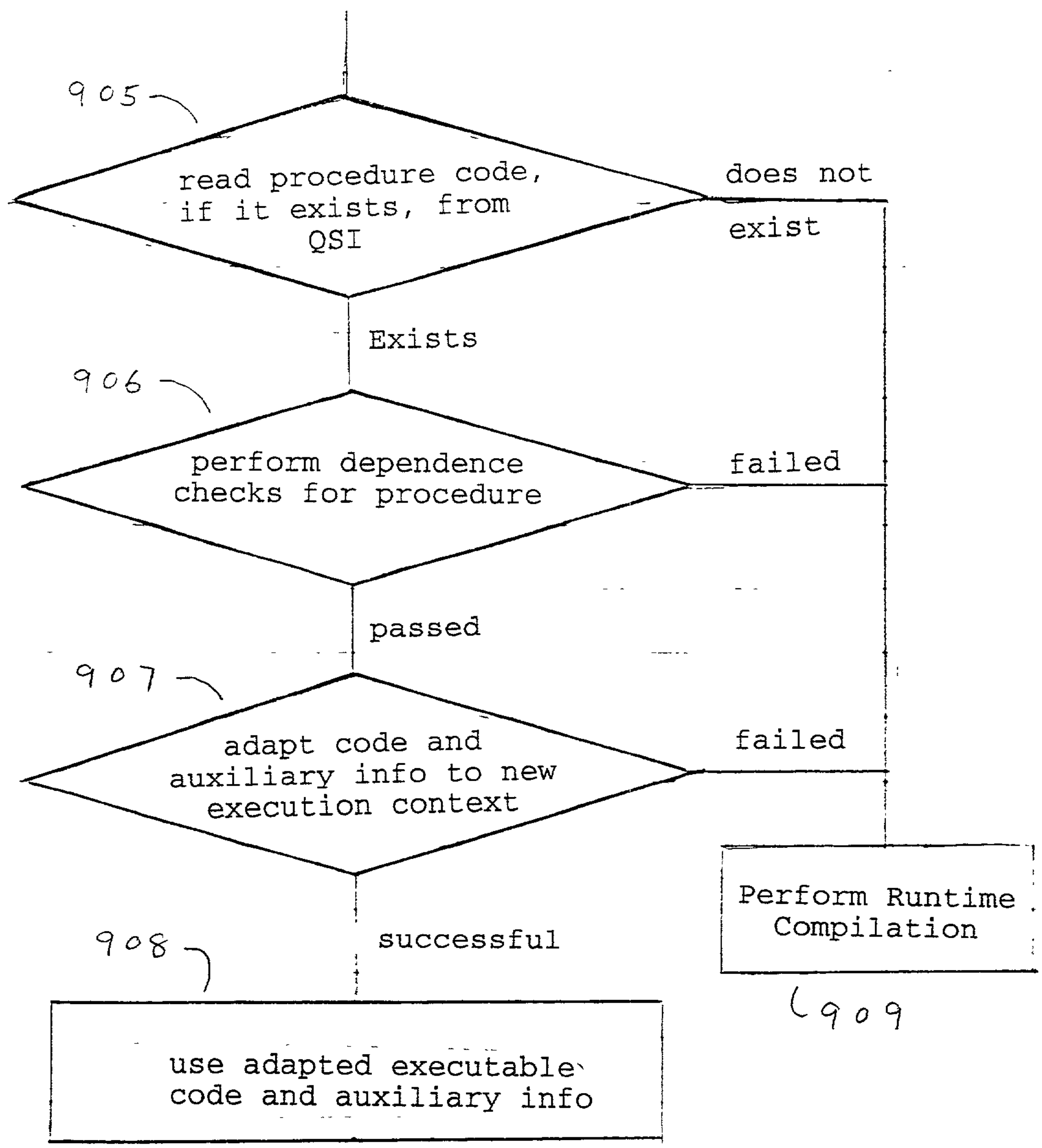
905
read procedure code, if it exists, from QSI
does not exist
Exists
906
perform dependence checks for procedure
failed
passed
907
adapt code and auxiliary info to new execution context
failed
successful
908
use adapted executable code and auxiliary info
Perform Runtime Compilation
909

Fig. 10

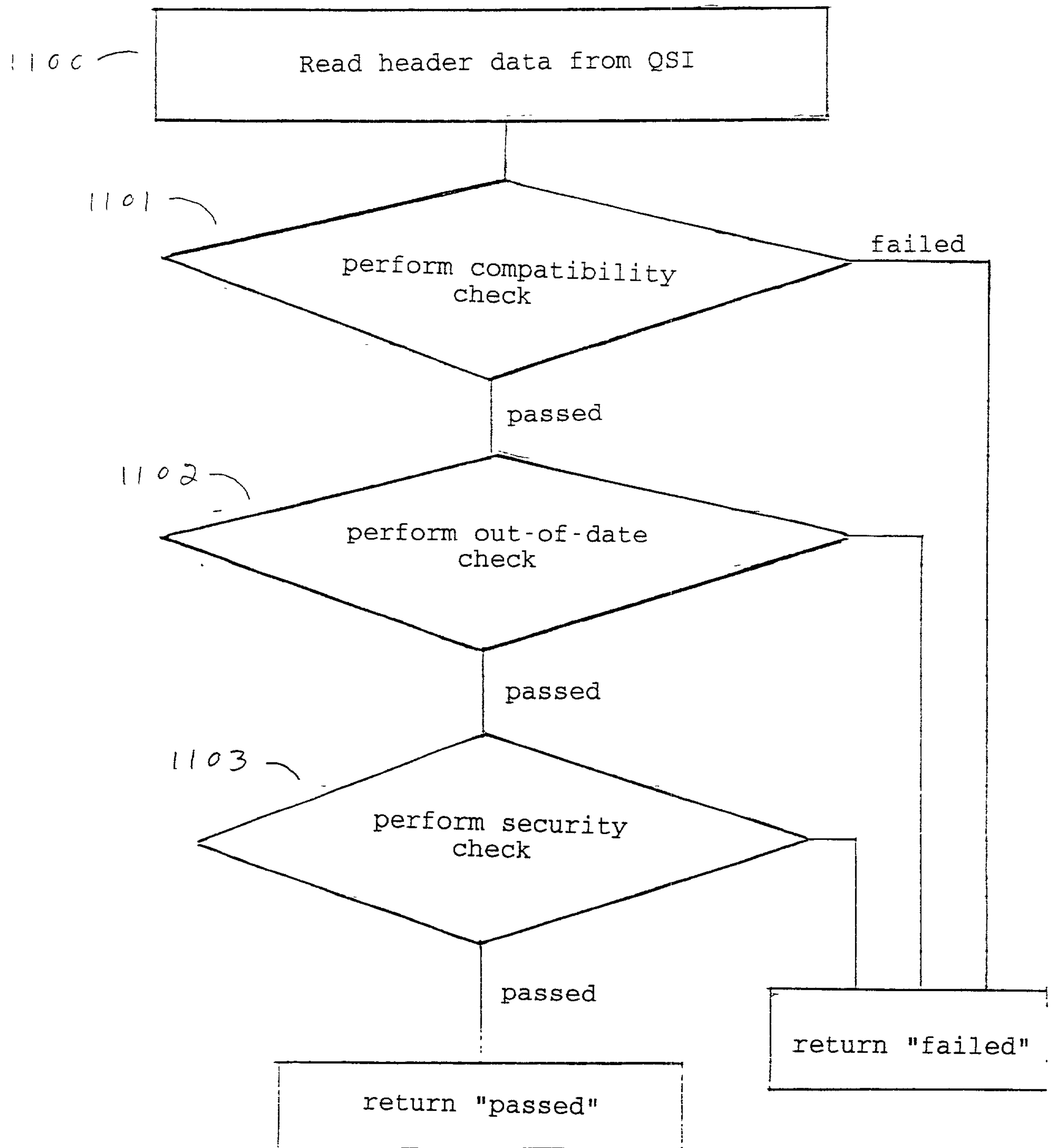
1100
Read header data from QSI
1101
perform compatibility check
failed
passed
1102
perform out-of-date check
passed
1103
perform security check
passed
return "passed"
return "failed"

Fig. 11

```
1200:          Adapt(code, auxiliary_info, ANNOT) {
1201:            for each item <I, T, S> in list of adaptation
annotations ANNOT do
1202:                locate the instruction or auxiliary info
corresponding to I ;
1203:                locate old information using T and replace
by new information computed            using S;
1204:                if (step 1003 failed) then return indicating
failure of the adaptation;
1205:               if (added extra instructions in step 1003)
then modify auxiliary information            such as
exception tables, garbage collection maps, if necessary.
1205:            }
1206:          }
```

Figure 12: Adapting code and auxiliary information to new execution context.

SSM&P Docket No.: 13675
IBM Docket No.:YOR9-2000-0365US1

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION

As a below named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below next to my name:

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled: METHOD FOR COMPILING PROGRAM COMPONENTS IN A MIXED STATIC AND DYNAMIC ENVIRONMENT

the specification of which (check one)

_x_ is attached hereto.

___ was filed on ____________________ as United States Application Number ____________________

or PCT International Application Number ________________

and was amended on ____________________ (if applicable)

I hereby state that I have reviewed and understand the contents of the above identified specification, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose information which is material to the patentability of this application in accordance with Title 37, Code of Federal Regulations, Section 1.56.

I hereby claim foreign priority benefits under Title 35, United States Code, §119(a)-(d) or §365(b) of any foreign application(s) for patent or inventor's certificate, or §365(a) of any PCT International application which designated at least one country other than the United States, listed below and have also identified below, by checking the box, any foreign application for patent or inventor's certificate, or PCT International application, having a filing date before that of the application on which priority is claimed:

Prior Foreign Application(s)			Priority Claimed
(Number)	(Country)	(Day/Month/Year Filed)	___ Yes ___ No
(Number)	(Country)	(Day/Month/Year Filed)	___ Yes ___ No
(Number)	(Country)	(Day/Month/Year Filed)	___ Yes ___ No

I hereby claim the benefit under 35 U.S.C. §119(e) of any United States provisional application(s) listed below.

(Application Number)	(Filing Date)
(Application Number)	(Filing Date)

I hereby claim the benefit under 35 U.S.C. §120 of any United States Application(s), or §365(c) of any PCT International application designating the United States, listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States, or PCT International application in the manner provided by the first paragraph of 35 U.S.C. §112, I acknowledge the duty to disclose information material to the patentability of this application as defined in 37 CFR §1.56 which occurred between the filing date of the prior application and the national or PCT international filing date of this application:

(Application Serial No.)	(Filing Date)	(Status) (patented, pending, abandoned)
(Application Serial No.)	(Filing Date)	(Status) (patented, pending, abandoned)

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that willful false statements may jeopardize the validity of the application or any patent issued thereon.

POWER OF ATTORNEY: As a named inventor I hereby appoint the following attorney(s) and/or agent(s) to prosecute this application and transact all business in the Patent and Trademark Office connected therewith (list name and registration number).

Manny W. Schecter (Reg. 31,722), Lauren C. Bruzzone (Reg. No. 35,802), Christopher A. Hughes (Reg. 26,914), Edward A. Pennington (Reg. 32,588), John E. Hoel (Reg. 26,279), Joseph C. Redmond, Jr. (Reg. 18,753), Douglas W. Cameron (Reg. No. 31,596), Wayne L. Ellenbogen (Reg. No. 43,602), Stephen C. Kaufman (Reg. No. 29,551), Daniel P. Morris (Reg. No. 32,053), Louis J. Percello (Reg. No. 33,206), Jay P. Sbrollini (Reg. No. 36,266), David M. Shofi (Reg. No. 39,835), Robert M. Trepp (Reg. No. 25,933), Paul J. Otterstedt (Reg. No. 37,411) and Louis P. Herzberg (Reg. No. 41,500).

JUL 21 '00 13:14

BSM&P Docket No.: 13675
IBM Docket No.:YOR9-2000-0365US1

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION

Send Correspondence to: Richard L. Catania, Scully, Scott, Murphy & Presser

400 Garden City Plaza, Garden City, New York 11530

Direct Telephone Calls to: (name and telephone number) Richard L. Catania, (516) 742-4343

Rajesh Bordawekar
Full name of sole or first inventor

RBordawekar 7/21/00
Inventor's Signature Date

1870 Baldwin Road #41, Yorktown Heights, New York 10598
Residence

India
Citizenship

Same as residence
Post Office Address

Manish Gupta
Full name of second joint inventor, if any

Manish Gupta 7/21/00
Inventor's signature Date

32 Huntington Circle, Peekskill, New York 10566
Residence

India
Citizenship

Same as residence
Post Office Address

Samuel Pratt Midkiff
Full name of third joint inventor, if any

Sam Pratt Midkiff 7/21/2000
Inventor's Signature Date

68 Danebury Downs, Upper Saddle River, New Jersey 07458-1536
Residence

United States of America
Citizenship

Same as residence
Post Office Address

SSM&P Docket No.: 13675
IBM Docket No.:YOR9-2000-0365US1

DECLARATION AND POWER OF ATTORNEY FOR PATENT APPLICATION

Mauricio J. Serrano
Full name of fourth joint inventor, if any

July 21/2000

Inventor's signature — Date

423 Sautner Drive, San Jose, California 95123
Residence

Colombia
Citizenship

Same as residence
Post Office Address

09624574 .072400